

Software Engineering in the Age of Delegation

James C. Davis, Steve France
Purdue University
{davisjam, sfrance1}@purdue.edu

Ayaan M. Kazerouni
Cal Poly
ayaank@calpoly.edu

Ayatallah M.M Ahmed, Lexy Arinze, Kirsten A. Davis
Purdue University
{ahmed486, larinze, kad}@purdue.edu

Abstract—Coding agents increasingly perform software implementation, challenging what software engineering education should teach and assess. We propose defining engineering as exercising informed control over consequential systems while accepting responsibility for their outcomes. This definition shifts educational priorities from performing implementation to exercising judgment over consequential decisions and governing delegated work. We operationalize these priorities in a redesigned undergraduate software engineering course enrolling 115 students at a large U.S. research-intensive institution. Students learn to make and defend engineering decisions through process, requirements, specification, architecture, design, and validation, then use models and controls to govern delegation. An open-ended, regulation-driven project requires teams to negotiate shared specifications and independently implement and cross-test their systems, while oral examinations assess individual engineering judgment. We describe the course’s initial offering and identify open questions about implementation expertise and the assessment of engineering competence.

Index Terms—Software engineering education, generative AI, engineering judgment & responsibility, human-AI collaboration

I. INTRODUCTION

Generative AI and increasingly capable coding agents have prompted calls to reconsider software engineering practice, the role of human engineers, and the computing curriculum itself [1], [2], [3]. What does it mean to engineer a software system when much of its fabrication can be delegated? Engineering has always involved delegation; agents greatly expand its scope. In business [4] and military [5] settings, leaders have long delegated execution while retaining responsibility for outcomes. Agents now enable individual engineers to do the same, delegating fabrication without relinquishing engineering responsibility. **We propose that software engineering education should train students to exercise informed control over delegated engineering work.**

The need for such control is increasingly apparent in industry. Feng *et al.* found that experienced engineers maintained control through deliberate delegation, while junior engineers struggled with overreliance on agents, particularly on unfamiliar tasks [6]. Anecdotally, a staff software engineer at SoFi described a pattern he had observed and heard from colleagues at Amazon, Meta, Google, and Microsoft: “code gets merged, but nobody is completely sure what the code does anymore” [7]. The 2026 State of Devs survey suggests the scale of this transition: respondents estimated a median of 63% of their code was AI-generated, while 45% identified negative cognitive impacts among their leading concerns [8].

These industrial developments and our own experience motivated us to redesign our undergraduate software engineering course. The lead instructor spent six months engaged in full-time agentic software development. He produced a three-million-line software system now in use at our institution, while directly authoring or inspecting almost none of its code. This experience led us to recognize the relevance of a systems-theoretic conception of engineering as control, and to develop practical mechanisms for exercising that control over delegated work. §III shows the developed educational approach; §IV illustrates its implementation through a semester project, oral examinations, and an agent-mediated modeling laboratory.

In summary, we contribute:

- **A responsibility-and-control framing** of software engineering that distinguishes delegable work from enduring engineering responsibility.
- **A software engineering course design** emphasizing engineering judgment and informed control when implementation is delegated.

II. BACKGROUND

A. From Generative AI to Agentic Engineering

Advances in generative AI [13] have enabled engineers to delegate complex software engineering work to computational agents [14], [15], [16]. We use *agents* to mean computational systems that perform delegated work requiring open-ended reasoning and action, and *fabrication* to mean the production or modification of engineering artifacts, including code, specifications, tests, and models. The declining cost of fabrication shifts engineering attention toward deciding what should be built, establishing obligations, evaluating evidence, and governing the resulting system [17].

Agents increasingly perform work throughout the software lifecycle [18], while software engineering education has traditionally emphasized students performing that work themselves [19]. Industry is still determining what this means for practice; educators should not simply follow. It is our responsibility to interpret these changes and prepare the next generation of engineers, at times in disagreement with current practice. Delegation instead requires engineers to establish conditions for acceptable results and retain the ability to evaluate and intervene.

B. Engineering: Problem Solving or Control?

These developments raise a more fundamental question: what is engineering, independent of the tools and techniques

TABLE I
TWO PERSPECTIVES ON ENGINEERING. THE CHOICE DETERMINES WHAT ENGINEERING EDUCATION SHOULD PRIORITIZE.

Problem-Solving Perspective	Informed-Control Perspective (advocated in this work)
“ <i>Engineering</i> : the profession in which a knowledge of the mathematical or physical sciences gained by study, experience and practice is applied with <i>judgement</i> to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind.” —ABET, quoted in [9]	The discipline of exercising <i>informed control</i> [10], [11] over consequential systems and <i>accepting responsibility</i> for their outcomes [12]. Control requires sufficient command to understand and direct the system, evaluate evidence, recognize failed assumptions, and intervene. Implementation may be delegated to humans or to machine intelligence. —Our synthesis

available to engineers? We propose defining engineering through the responsibilities practitioners retain as tools evolve.

Engineering is conventionally characterized as the application of scientific knowledge and technical judgment to solve problems. Professional codes of ethics establish engineers’ responsibility for the consequences of their work [20]. Accreditation criteria likewise emphasize solving complex engineering problems alongside ethical judgment and professional responsibility [21]. Yet problem-solving remains central to how engineering competence is described [22].

As AI agents acquire engineering capabilities, problem-solving becomes an insufficient definition. Agents, too, can analyze problems, develop designs, implement solutions, and evaluate results. What then distinguishes an engineer from an agent? We argue that the distinction lies not in the activities performed, but in responsibility for their consequences.

Safety engineering provides a useful foundation for this view. Leveson’s systems-theoretic approach treats safety as a control problem: enforcing constraints on system behavior through appropriate control structures, feedback, and intervention [10], [11]. From this perspective, the engineer must design, understand, and direct a (largely automated) system, evaluate its behavior, and intervene when necessary. Engineering entails informed control over consequential systems and responsibility for their outcomes (Table I). **Responsibility requires sufficient understanding and authority to evaluate and influence consequential behavior.**

Practical motivation. In our previous course offering, students were given a detailed specification for a software package registry, *e.g.*, npm or Hugging Face. One team completed the approximately 30-KLOC project over a weekend and earned the highest score, describing how they used a coding agent with little engineering judgment or understanding. Successful implementation did not demonstrate competence.

III. TEACHING SOFTWARE ENGINEERING AS DELEGATION

Our redesigned undergraduate software engineering course emphasizes two capabilities: engineering judgment and informed control. Judgment concerns deciding what matters and why. Control concerns ensuring that delegated work satisfies engineering obligations. Traditional software engineering topics remain, but we teach them through the lens of judgment and control. Later units explicitly address delegation to agents. The course concludes with software in operation (Table II).

A. Teaching Engineering Judgment, Not Implementation

Building on project-based and inductive learning [23], we teach conventional software engineering topics through the decisions engineers must make rather than the artifacts they produce. Students identify alternatives, distinguish their relevant properties, and establish evidence to justify decisions about systems they may not implement themselves. Three examples illustrate this emphasis.

Specification concerns the allocation of responsibility and freedom. Students distinguish obligations from fabrication choices, deciding which unresolved choices to constrain, delegate, or investigate further. A specification defines acceptable implementations rather than prescribing a particular one.

Architecture and design concern allocating system responsibilities and selecting mechanisms. Students compare alternatives based on dependencies, critical paths, failure propagation, performance, and maintainability. When agents can readily implement competing designs, students can investigate more alternatives before selecting and justifying a design against the system’s requirements.

Validation concerns the evidence needed to accept an implementation. Students identify claims about the system, select tests and analyses that could support them, and judge whether the evidence is sufficient. Traditionally, developers could draw on their familiarity with code they had written. When implementation is delegated, students must evaluate evidence without assuming such familiarity.

Students implement substantial software throughout the project, often with coding agents; classroom instruction emphasizes their decisions, delegation, and evaluation of evidence. §IV describes how we assess these capabilities.

B. Teaching Control: Models, Alignment, Degrees of Freedom

Delegation requires sufficient knowledge to direct and assess work without performing it personally. Because agents resolve unspecified choices, engineers must distinguish decisions they can safely delegate from those requiring explicit constraints or independent evidence. We teach this through models, alignment, and degrees of freedom.

a) *Models*: An engineering model is a purposeful approximation that preserves the properties needed to answer an engineering question while omitting the rest. Languages such as UML and SysML support reasoning about structure, behavior, and constraints [24], [25]. Students identify an engineering question, determine which properties matter, and construct a

TABLE II
SYLLABUS ORGANIZED AROUND ENGINEERING JUDGMENT AND CONTROL UNDER DELEGATION.

Unit	Engineering capability	Course application
Engineering foundations	Distinguish engineering decisions from fabrication; establish team processes	Organize project teams and responsibilities
Requirements	Distinguish stakeholder desires from engineering commitments	Interpret regulations and establish requirements
Specification	Allocate responsibility; constrain consequential behavior while preserving design freedom	Negotiate shared specifications
Architecture & design	Compare mechanisms, dependencies, and tradeoffs using evidence	Defend architecture and design decisions
Validation	Determine what evidence supports engineering claims	Design validation and cross-team testing
Delegation	Allocate authority and establish conditions for acceptable results	Delegate implementation to coding agents
Modeling	Represent consequential properties without prescribing implementation	Construct and analyze engineering models
Alignment	Make obligations independently checkable or enforceable	Govern agent-produced artifacts
Failure-aware engineering	Diagnose failures and strengthen models and controls	Convert recurring failures into durable controls
Software in the world	Maintain informed control as systems operate and evolve	Reason about deployed systems and changes

model sufficient to answer it, making important system properties inspectable without prescribing implementation. Prior work has used interactive modeling environments and model-based systems engineering to develop modeling skills and systems thinking [26], [27]. We extend this educational purpose to delegation, using models to constrain delegated work. Formal modeling remains uncommon in everyday software engineering [28], making it difficult to justify as a central activity in introductory courses. Agents may change this calculus: they reduced the effort of constructing and manipulating models, while delegated implementation increased the value of independently checking system properties. We therefore integrate modeling into the engineering process rather than treating it as a specialized technique (§IV-C).

b) Alignment: Models express obligations but do not enforce them. Alignment uses tests, static analyses, and other checks to evaluate artifacts independently of an agent’s explanations. Yet verification establishes only what follows from the supplied model and properties: an agent may produce a model that passes every check while misrepresenting the intended system, omitting consequential assumptions, or preventing useful behavior. Students must therefore judge whether models and checks express the right obligations.

c) Degrees of freedom: Every engineering decision limits the set of possible implementations. Models specify selected properties while deliberately leaving others unspecified. Delegation allocates these remaining degrees of freedom: the engineer determines what must hold while allowing the agent to choose how to satisfy it. Over-specification needlessly restricts implementation choices; under-specification leaves important system behavior unspecified. Deciding which implementation choices must be constrained and which can safely remain open is itself an exercise of engineering judgment.

IV. COURSE ACTIVITIES AND ASSESSMENT

We are implementing this approach in an undergraduate software engineering course with 115 students. Three activities foreground judgment and control, each doing different work. A semester project organized around shared specifications and reciprocal testing *elicits* engineering decisions and records them in inspectable artifacts. Individual oral examinations *assess* whether those decisions are understood by each student rather than only by the team. A modeling laboratory that

separates agent assistance from semantic authority *examines the limits* of the evidence automated analysis produces.

The student activities and observations reported in this section were conducted under a protocol approved by our Institutional Review Board (IRB).

A. Semester Project

a) Project organization: The semester project asks students to translate regulatory obligations into requirements and specifications, resolving ambiguous terms while retaining implementation freedom. LLMs make hundreds of pages of regulatory text accessible; students must still exercise engineering judgment to determine what their systems should do. They select from four regulatory contexts:

- 1) **ADA Title II** [29]: document accessibility.
- 2) **EU AI Act** [30]: AI transparency.
- 3) **EU Cyber Resilience Act** [31]: product security.
- 4) **SEC Disclosure Rules** [?]: incident reporting.

Teams of 3–5 students interpret the applicable regulation, select a tractable subset of its obligations, and develop requirements, specifications, designs, implementations, and evidence of compliance. The instructor supplies neither the engineering problem nor its specification: students must determine what to build and how. The project has seven phases: (0) form teams and select regulations; (1) investigate regulations, stakeholders, and candidate projects; (2) establish requirements, interfaces, and success criteria; (3–5) complete team-defined development milestones, including testing and validation; and (6) demonstrate prototypes and supporting evidence.

Teams are organized into *pods* according to the regulation they address. Within each pod, teams negotiate a shared specification before implementing independently and cross-testing one another’s systems. Regulatory ambiguity requires students to determine what must be specified. Pods make testing these decisions scalable: independent implementations and cross-testing expose differences that agreement on a specification may conceal, without instructor-provided reference implementations. Students practice establishing obligations that remain meaningful when others carry out the work.

At the time of writing, teams have completed the first development milestone (Phase 3).

b) *Preliminary observations:* We examined the submissions for Phases 1–3 (30 teams) for evidence of students’ problem investigations, specifications, and design decisions.

The submissions illustrate how students translated regulatory uncertainty into explicit engineering decisions. In an ADA Title II project, students initially recognized that they “*cannot assume that accessibility can be handled as a completely automated pass or fail process.*” Their subsequent specification distinguished two automatically repairable checks from six requiring human review, prohibited unsupported claims of ADA compliance, and established acceptance tests for these restrictions. In a Cyber Resilience Act project, students disagreed about when a manufacturer legally “becomes aware” of an exploited vulnerability. Rather than resolve the legal ambiguity themselves, they required the timestamp to be supplied by a user, preserving deterministic deadline calculations without delegating the legal judgment to software.

The reports also exposed shortcomings in engineering judgment. One pod declared all disagreements resolved, although a team’s risk analysis identified unresolved decisions about supported formats, checks, and thresholds. One team’s design notes (for an EU AI Act project) state that without a watermark, the program “*cannot conclude about the file’s origins,*” yet the team specified no requirement or acceptance criterion for human review. Across these cases, the allocation of authority between people and software, and its translation into requirements and acceptance criteria, became inspectable—including where that translation was incomplete (§III).

Extensive delegation to coding agents did not remove the need to establish shared engineering obligations. Phase 3 reports often describe how cross-team testing exposed unresolved choices in both product behavior and testing procedures. One team reported that “*The two teams’ tests disagreed [on precedence of error code to apply].*” Another reported that “*Both teams assumed exclusive use of the system under test*” and failed when tests ran concurrently. Pods thus made the consequences of underspecification visible.

B. Individual Oral Examinations

a) *Assessment design:* Project success does not establish individual competence: students may rely on teammates’ decisions or agents’ implementations without understanding how to evaluate the resulting system.

We will administer two rounds of individual oral examinations in lieu of midterm and final exams, lasting approximately five minutes each. Oral examinations have been studied as a means of assessing individual understanding [32], and are recommended in response to generative AI [33]. Students explain and defend consequential decisions about their project’s requirements, specification, design, and validation, regardless of who—or what—produced the implementation.

b) *Preliminary observations:* Thus far we have completed 50 of the first-round oral examinations, of approximately five minutes each. We observed that technical follow-up questions exposed substantial differences in students’ understanding of their projects. These anecdotal observations

suggest that brief oral examinations can help distinguish informed delegation from reliance on agent-produced work that students cannot adequately explain or evaluate.

C. Modeling Laboratory

a) *Workbench design:* We developed the Workbench to introduce students to engineering with semantically defined models. Following the computer-aided software engineering tradition [34], it draws on SysML v2 and KerML [25], [35] and contemporary model-driven environments such as BESSER [36]. It supports typed entities, state machines, synchronized events, and dimensioned quantities.

For classroom accessibility, we provide a static client-side application requiring no server, account, or installation, addressing usability barriers identified in modeling education [37], [38]. Students construct and analyze models through a graphical interface or an existing coding agent connected to the browser through the Chrome DevTools Protocol. This provides natural-language interaction without requiring students to master the modeling notation. Crucially, interaction is separate from semantic authority: the agent translates intent into model operations and reports results, but the Workbench’s semantics determine what an analysis concludes.

b) *Laboratory exercise:* Students use the agent-assisted workbench to investigate two design problems sharing one structure. In the first, concurrent threads must share a limited memory budget; in the second, a message-processing system must avoid losing acknowledged work after a failure. Each begins with an unsafe design. Students compare a conservative solution that restricts behavior with a refined solution permitting greater concurrency or earlier acknowledgment. They use the model checker to examine both whether undesirable states are excluded and whether the system can still perform useful work: eliminating undesirable behavior is not sufficient if the solution also eliminates desirable behavior. In the first activity, for example, threads requiring up to 20, 24, and 27 KB of working memory share a 48 KB budget; serializing allocations is safe but needlessly restrictive, while a refined policy can permit the 20 KB and 27 KB allocations to coexist—though not the 24 KB and 27 KB pair. The exercise therefore requires students to distinguish a passing verification result from an adequate engineering design. For each activity, students report two distinct design approaches, articulate the properties they established for each, and reflect on their experience designing with an agent and its implications for software engineering.

c) *Preliminary observations:* We analyzed written reports from the 64 students who completed the modeling laboratory. Codes were developed inductively and applied consistently across the reports using LLM assistance.

In both activities, 59% explicitly reported checking not only that unsafe states were excluded but that the system could still do useful work—the distinction the laboratory exists to teach, and one that a correct model alone does not establish. Students also described agent assistance as accelerating model construction while leaving important decisions and verification responsibilities with them: 48% mentioned speed, 53%

described retained human responsibility, and 33% expressed both. One student explained that “*the agent could make a valid model that did not match the actual constraint I was trying to preserve.*” Difficulties were equally concrete: some reported an agent changing a system parameter to satisfy a failing check, and others verification results silently carried over after a revised model failed to load. These failures illustrate why students must inspect both the models agents construct and the evidence produced by verification.

D. Formal Evaluation Plan

We will evaluate students’ ability to identify consequential decisions, allocate responsibility, and assess evidence about delegated work. Project artifacts will show traceability from regulatory obligations to requirements, treatment of unresolved decisions, and adequacy of acceptance criteria. Cross-testing will examine whether independently developed implementations satisfy shared specifications. Oral examinations will assess students’ ability to explain and defend their engineering decisions. We will compare project and individual assessments to identify recurring failures of judgment and control.

V. DISCUSSION

The course illustrates how a definition of engineering grounded in responsibility and informed control (Table I) can guide educational design, but leaves important questions about the competencies engineers must retain. This framing emphasizes enduring responsibilities: delegating, evaluating evidence, recognizing failed assumptions, and intervening.

Must engineers retain implementation expertise? Under our informed-control framing of engineering, engineers must be competent in requirements, specification, architecture, validation, and other activities needed to direct and assess the production of software. Implementation—writing the code—is the activity most plausibly fully delegable to agents. Must engineers still learn to code? Historically, software abstractions have leaked: engineers sometimes need to reason about underlying mechanisms to diagnose failures and assess consequences [39]. Bainbridge’s ironies of automation identify a related danger: automation can remove opportunities to develop and maintain the expertise needed for intervention [40].

Consider a system that faithfully implements its specification yet fails because important behavior was left unspecified. Diagnosing such a failure requires reasoning in *machine-space*—examining the implemented system’s behavior—and translating that understanding into *model-space*—refining the engineering models that govern subsequent implementations. Thus, implementation expertise may remain necessary not primarily to produce artifacts, but to recognize when the models governing their production are inadequate. The modeling laboratory suggests the same distinction: being able to generate a model does not establish the ability to understand its assumptions or judge its results—the ability to produce an artifact and the ability to evaluate it are distinct competencies. Engineering education must therefore determine how much implementation expertise students need to exercise informed

control, and how to develop and sustain that expertise as implementation becomes increasingly automated.

What should count as engineering competence? Producing working software has long served as a proxy for competence. This proxy is now weakened by agentic development. Assessment must examine students’ requirements, decisions, delegation, and supporting evidence. Projects and oral examinations can assess these abilities, but must distinguish between work students performed themselves and work they directed and evaluated. Both are plausible targets for assessment.

VI. RELATED WORK

Recent work reconsiders human–agent division of labor across engineering processes and the software lifecycle [18], [1], [41], model-based governance and durable controls [42], [43], and human agency and inspection [2], [44]. Contemporary accounts of agentic software engineering describe a shift from code production toward specification, verification, and human judgment [45], [46]. These accounts identify where human expertise is currently needed. We instead ask what remains when agents can perform these activities too: the enduring responsibility to direct engineering work, evaluate its consequences, and intervene when necessary.

Research on AI in computing education has examined opportunities and risks for student learning [47]. In software engineering education, prior work explores incorporating AI into existing curricula [48], integrating it across design, implementation, and testing [49], and assessing AI-assisted development through process and reflection rather than code alone [50]. Other work proposes frameworks for responsible AI integration [51]. These approaches recognize that responsibility remains with the developer even as AI assumes more engineering work. Judgment-centered education further emphasizes verification, accountability, and evidence of individual understanding [52]. We share this emphasis on judgment, but ask what that responsibility requires students to know and do, and use the answer to reorganize software engineering education around modeling, delegation, and alignment.

Modeling is central to education and practice in other engineering disciplines, where physical changes are costly. It supports prediction and design exploration [53], [54]. We draw on this tradition to counter premature code fixation: students reason about system properties and assumptions before committing to implementation. Agentic development extends these practices to delegated work, using models to direct implementation and evaluate its results.

VII. CONCLUSION

Software engineering education should be organized around the responsibilities engineers retain, not the activities that agents cannot yet perform. We propose developing two complementary capabilities: judgment over consequential decisions and informed control over delegated fabrication. Our course shows how these objectives can organize undergraduate instruction. Future research must determine which models and controls best support increasingly complex delegation.

Materials: The open course materials include a textbook on decision-making in software engineering, lecture notes, slides, and the course project. See <https://davisjam.github.io/model-based-agentic-software-engineering/teach/index.html>.

Acknowledgments. This work was supported by NSF award #2452533.

REFERENCES

- [1] A. E. Hassan, H. Li *et al.*, “Agentic Software Engineering: Foundational Pillars and a Research Roadmap,” *arXiv:2509.06216*, 2025.
- [2] S. Abrahão, J. Grundy *et al.*, “Software Engineering by and for Humans in an AI Era,” *ACM TOSEM*, vol. 34, no. 5, 2025.
- [3] C. N. Sekharan and G. K. Thiruvathukal, “Now’s the Time: Computer Science Must Evolve to Emphasize Software and Systems Engineering With AI,” *Computer*, vol. 59, no. 8, pp. 117–121, 2026.
- [4] R. Simons, “Control in an age of empowerment,” *Harvard Business Review*, vol. 73, no. 2, pp. 80–88, 1995.
- [5] Department of the Army, *ADP 6-0, Mission Command: Command and Control of Army Forces*, U.S. Department of the Army, 2019.
- [6] D. Feng, B. Yun *et al.*, “From junior to senior: Allocating agency and navigating professional growth in agentic AI-mediated software engineering,” in *CHI*. ACM, 2026.
- [7] C. Lin, “LinkedIn post on AI-generated code and AI-assisted review,” LinkedIn, 2026, accessed: 2026-10-08. [Online]. Available: <https://www.linkedin.com/feed/update/urn:li:activity:7506430566133723136/>
- [8] Devographics, “State of Devs 2026,” Online survey, 2026, accessed: 2026-10-09. [Online]. Available: <https://2026.stateofdevs.com/en-US/>
- [9] S. Reisman, “Engineering? No Way,” *Computer*, 2025.
- [10] N. G. Leveson, *Safeware: system safety and computers*. Addison-Wesley, 1995.
- [11] —, *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2012.
- [12] G. Bugliarello, “The Social Function of Engineering: A Current Assessment,” in *Engineering as a Social Enterprise*, H. E. Sladovich, Ed. Washington, DC: National Academy Press, 1991.
- [13] X. Hou, Y. Zhao *et al.*, “Large language models for software engineering: A systematic literature review,” *ACM TOSEM*, 2024.
- [14] K. Li, A. K. Hopkins *et al.*, “Emergent World Representations: Exploring a Sequence Model Trained on a Synthetic Task,” in *ICLR*, 2023.
- [15] C. Jin and M. Rinard, “Emergent Representations of Program Semantics in Language Models Trained on Programs,” in *ICML*, 2024.
- [16] Y. Zhang, H. Ruan *et al.*, “Autocoderover: Autonomous program improvement,” in *ISSTA*, 2024.
- [17] E. M. Goldratt and J. Cox, *The Goal: A Process of Ongoing Improvement*. North River Press, 1984.
- [18] A. Roychoudhury, “Agentic AI for Software: Thoughts from Software Engineering Community,” *arXiv:2508.17343*, 2025.
- [19] C. Sengul, R. Neykova *et al.*, “Software engineering education in the era of conversational AI: Current trends and future directions,” *Frontiers in Artificial Intelligence*, vol. 7, 2024.
- [20] Association for Computing Machinery, “ACM Code of Ethics and Professional Conduct,” ACM, 2018. [Online]. Available: <https://www.acm.org/code-of-ethics>
- [21] “Criteria for Accrediting Engineering Programs, 2026–2027,” Engineering Accreditation Commission, ABET, 2026, accessed 2026-10-09. [Online]. Available: <https://www.abet.org/accreditation/accreditation-criteria/>
- [22] G. L. Downey, “The local engineer: Normative holism in engineering formation,” in *Engineering, Development and Philosophy: American, Chinese and European Perspectives*. Springer, 2012.
- [23] M. J. Prince and R. M. Felder, “Inductive Teaching and Learning Methods: Definitions, Comparisons, and Research Bases,” *Journal of Engineering Education*, vol. 95, no. 2, 2006.
- [24] Object Management Group, “OMG Unified Modeling Language (UML), Version 2.5.1,” Object Management Group, OMG Specification formal/17-12-05, 2017. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1>
- [25] —, “OMG Systems Modeling Language (SysML), Version 2.0,” Object Management Group, OMG Specification formal/26-03-02, 2025. [Online]. Available: <https://www.omg.org/spec/SysML/2.0>
- [26] S. Krusche, N. von Frankenberg *et al.*, “An interactive learning method to engage students in modeling,” in *ICSE-SEET*. ACM, 2020, pp. 12–22.
- [27] K. Chandra, S. Kraemer *et al.*, “Integrating model-based systems engineering and systems thinking skills in engineering courses,” in *ASEE*. ASEE, 2024.
- [28] M. Petre, “UML in practice,” in *ICSE*. IEEE, 2013, pp. 722–731.
- [29] U.S. Department of Justice, “Nondiscrimination on the Basis of Disability; Accessibility of Web Information and Services of State and Local Government Entities,” Federal Register, 89 FR 31320, 2024, final rule, April 24, 2024.
- [30] European Parliament and Council of the European Union, “Regulation (EU) 2024/1689 Laying Down Harmonised Rules on Artificial Intelligence,” Official Journal of the European Union, 2024, Artificial Intelligence Act.
- [31] —, “Regulation (EU) 2024/2847 on Horizontal Cybersecurity Requirements for Products with Digital Elements,” Official Journal of the European Union, 2024, Cyber Resilience Act.
- [32] P. Ohmann and E. Novak, “A Multi-Institutional Assessment of Oral Exams in Software Courses,” in *SIGCSE*. ACM, 2025.
- [33] MIT Ad Hoc Committee on AI Use in Teaching, Learning, and Research Training, “AI and Education,” Massachusetts Institute of Technology, Tech. Rep., Aug. 2026. [Online]. Available: <https://aiandeducation.mit.edu/report/>
- [34] A. Fuggetta, “A classification of CASE technology,” *Computer*, 1993.
- [35] Object Management Group, “KerML: Kernel Modeling Language, Version 1.0,” Object Management Group, OMG Specification formal/26-03-01, 2025. [Online]. Available: <https://www.omg.org/spec/KerML/1.0>
- [36] I. Alfonso, A. Conrardy *et al.*, “Building BESSER: An open-source low-code platform,” in *Enterprise, Business-Process and Information Systems Modeling (BPMDS/EMMSAD 2024)*. Springer, 2024.
- [37] G. Liebel, O. Badreddin *et al.*, “Model driven software engineering in education: A multi-case study on perception of tools and UML,” in *CSEE&T*. IEEE, 2017, pp. 124–133.
- [38] S. Zschaler, T. C. Lethbridge *et al.*, “What do we need from modeling tools for teaching? a survey of the community of higher-education modeling teachers,” *SoSyM*, 2025.
- [39] J. Spolsky, “The Law of Leaky Abstractions,” 2002, Joel on Software. [Online]. Available: <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/>
- [40] L. Bainbridge, “Ironies of Automation,” *Automatica*, vol. 19, no. 6, 1983.
- [41] A. E. Hassan, G. A. Oliva *et al.*, “Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap,” *ACM TOSEM*, 2026.
- [42] J. C. Davis, K. Kalu *et al.*, “Model-Based Agentic Software Engineering,” *arXiv:2608.25174*, 2026.
- [43] J. C. Davis, P. C. Amusuo *et al.*, “Cheap Code, Costly Judgment: A Case Study on Governable Agentic Software Engineering,” *arXiv:2607.01087*, 2026.
- [44] M. Monperrus, “The end of code review: Coding agents supersede human inspection,” *arXiv:2606.13175*, 2026.
- [45] K. Tantithamthavorn, *Agentic Software Engineering: A Practical Guide for the AI-Native Engineer*, 2026, open-access textbook. [Online]. Available: <https://book.agentic-swe.dev/>
- [46] A. E. Hassan, *Agentic Software Engineering: The Future of Code*, 2026, online book. [Online]. Available: <https://agenticse-book.github.io/>
- [47] F. J. Agbo, C. Olivia *et al.*, “Computing education using generative artificial intelligence tools: A systematic literature review,” *Computers and Education Open*, vol. 9, 2025.
- [48] M. Vierhauser, I. Groher *et al.*, “Towards integrating emerging AI applications in SE education,” in *CSEE&T*, 2024.
- [49] N. Roy, O. Horiello *et al.*, “Integrating AI tools in advanced computer science curricula: A case study of course redesign,” in *CompEd*, 2025.
- [50] B. Fein, G. Fraser *et al.*, “AI-driven software development: A new course concept and assessment model for the era of large language models,” in *ICSE-SEET*, 2026, pp. 386–396.
- [51] V. Garousi, Z. Jafarov *et al.*, “Encouraging responsible GenAI use in software engineering education: A design-oriented model,” *JSS*, 2026.
- [52] Q. H. Mahmoud, “Judgment-centred software engineering education: A post-hype review and framework for AI-augmented learning,” 2026.
- [53] A. R. Carberry and A. F. McKenna, “Exploring student conceptions of modeling and modeling uses in engineering design,” *Journal of Engineering Education*, vol. 103, no. 1, pp. 77–91, 2014.
- [54] R. L. Nagel and M. R. Bohm, “On teaching functionality and functional modeling in an engineering curriculum,” in *IDETC/CIE*, 2011.